Brad Abrams

Designing Microsoft® .NET Class Libraries

June 2004

Lesson 4: Member Types

- After successfully completing this lesson, you will be able to:
  - Use the right kind of member in the right places
  - Understand the trade-offs in using one kind of member instead of another

Kinds of Members

- Classes and ValueTypes can have:
  - Constructors
  - Methods
  - Fields (data members)
  - Properties
  - Events

Constructors

- Do minimal work in the constructor
  - Only capture the parameters
  - Cost is delayed
- You can throw exceptions from constructors
- Be consistent in the ordering and naming of constructor parameters

Constructors: Implicit Constructors

- Many languages automatically add a public default constructor if you don't specify any
  - Abstract classes get a protected constructor
- These two code snippets are equivalent:

```
public class Foo {
}
```

```
public class Foo {
    public Foo () {}
}
```

Constructors: Implicit Constructors (cont.)

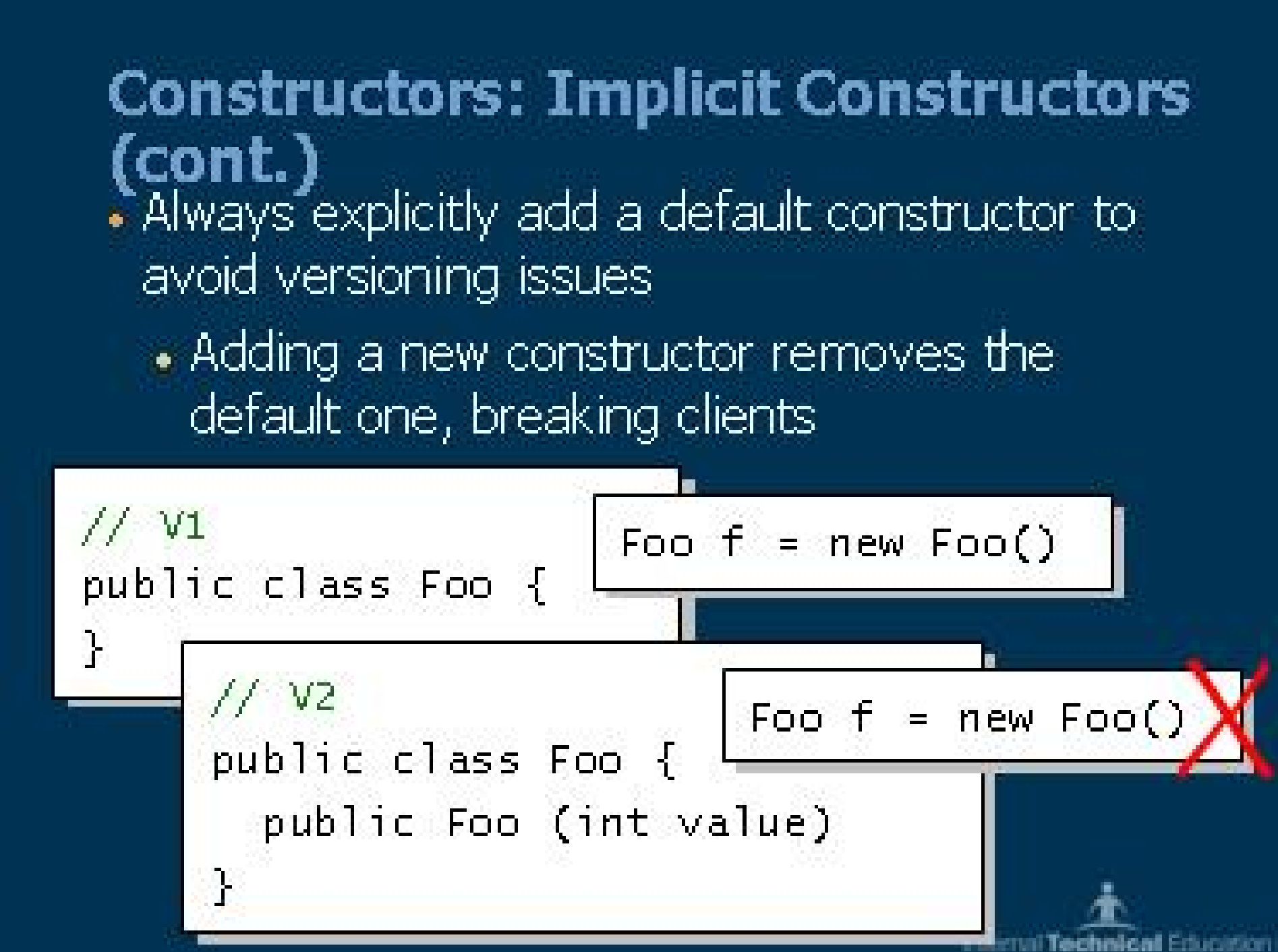
- Always explicitly add a default constructor to avoid versioning issues
  - Adding a new constructor removes the default one, breaking clients

```
// V1
public class Foo {
}
```

```
Foo f = new Foo()
```

```
// V2
public class Foo {
  public Foo (int value)
}
```

```
Foo f = new Foo()
```



Constructors and Properties

- A constructor's parameters are shortcuts for setting properties
  - No difference in semantics between these code snippets:

```
EventLog log = new EventLog();
log.MachineName = "kcwalina0";
log.Log = "Security";
```

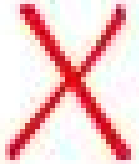
```
EventLog log = new EventLog("Security");
log.MachineName = "kcwalina0";
```

```
EventLog log = new EventLog("Security","kcwalina0");
```

Method Overloading: Semantics

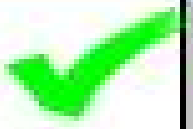
- Use overloading only when the overloads do semantically the same thing
  - Incorrect overload:

```
String.IndexOf(string value) {}
String.IndexOf(char[] anyOf) {}
```



  - Correct overload:

```
Convert.ToString(int value) {}
Convert.ToString(double value) {}
```



Method Overloading: Defaults

- Use appropriate default values
  - Simple method assumes default state
  - More complex methods indicate changes from the default state

```
MethodInfo Type.GetMethod (string name);
    //ignoreCase = false

MethodInfo Type.GetMethod (string name,
    boolean ignoreCase);
```

- Use a zeroed state for the default value (such as: 0, 0.0, false, "", *etc.*)

Method Overloading: Example

- Be consistent in the ordering and naming of method parameters
- Only the method with the most parameters should be virtual if needed

```
public class Foo {
   private const string defaultForA = "a default";
   private const int defaultForB = 42;
   public void Bar(){
      Bar(defaultForA, defaultForB);
   }
   public void Bar (string a){
      Bar(a, defaultForB);
   }
   public /*virtual*/ void Bar (string a, int b){
      // core implementation here
   }
}
```

Methods: Params

- Variable number of arguments (example: printf)
  - Use **params**

```
public static string Format(string format,
     params object[] args);
```

```
s.Format("my {0} dogs eat {1}.", 2, "puppy-chow");
```

  - Not used for in/out params (example: scanf)

Methods: Performance

- Allowing method in-lining by the JIT\NGen:
  - Minimize the use of virtual methods
  - Don't write really large methods
  - Don't have large numbers of locals

Fields

- Avoid publicly exposed instance fields
  - Properties offer more flexibility at minimal cost
    - JIT\NGEN inlines simple property access
    - Easy to add cache or delay creation in the future
- For static fields, do not include a prefix on a public field name
  - Example: 'g_' or 's_' to distinguish static vs. non-static fields

Const vs. Readonly

const

- Compile-time evaluation
- Stable across versions
- Always static

readonly

- Run-time evaluation
- Unstable across versions
- Static or instance

```
class Math {
  public const double Pi = 3.14;
}

class Color {
  public static readonly Color Red   = new Color(...);
  public static readonly Color Blue  = new Color(...);
  public static readonly Color Green = new Color(...);
}
```

Properties

- Smart fields
  - Calling syntax like fields
  - Flexibility of methods

```
public class Button: Control {
   private string text;
   public string Text {
      get {
         return text;
      }
      set {
         text = value;
         RepaintO;
      }
   }
}
```

```
Button b = new Button();
b.Text = "OK";
string s = b.Text;
```

Properties

- Use read-only properties where appropriate
- Do not use write-only properties
- Consider raising PropertyChanged events
- Property getters should be simple and therefore unlikely to throw exceptions
- Properties should not have dependencies on each other
  - Setting one property should not affect other properties
- Properties should be settable in any order

Properties

- Common to have read-only public access and protected write access

```
Public class MyClass {
    private string name;
    public string Name {
      get {
          return name;
      }
    }
    protected void SetName (string name)
    {
        this.name = name;
    }
}
```

Properties vs. Methods

- Do use a property:
  - If the member has a logical data member

```
string Name {get;}
```

```
string GetName()
```

```
Guid GetNext(){}
```

```
Guid Next {get;}
```

Properties vs. Methods (continued)

- Do use a method:
  - If the operation is a conversion, such as `ToString()`
  - If the getter has an observable side effect
  - If order of execution is important
  - If the method might not return immediately

Properties vs. Methods (3)

- Do use a method: (continued)
  - If the member returns an array

```
EmployeeList l = FillList();
for (int i = 0; i < l.Length; i++) {
    if (l.All[i] == x){...}
}
```

```
if (l.GetAll()[i]== x) {...}
```

- Creates a new snap shot of the array each time through the loop
- The `GetAll()` form makes this much clearer

Properties: Indexers

- Use if the logical backing store is an array
- Almost always object, int, or string indexed
- Rare to have multiple indices

```
public char this[int index] {get;}
```

```
String s = "foo";
Console.WriteLine (s[i]); // calls indexer
```

Events

- Defining an event

```
public delegate void EventHandler(object sender,
    EventArgs e);

public class Button: Control {
   public event EventHandler Click;

   protected void OnClick(EventArgs e) {
      if (Click != null)
         Click(this, e);
   }
}
```

Events

- Using an Event

```
void Initialize() {
    Button b = new Button(...);
    b.Click += new EventHandler(ButtonClick);
}

void ButtonClick(object sender, EventArgs e)
{
    MessageBox.Show("You pressed the button");
}
```

Events

- Terminology: Events are **raised**, not triggered or fired
- Name events with a verb
  - Example: Click, Paint, DrawItem, DropDown,
- Event handlers have void return type

```
public delegate void MouseEventHandler (
                                    object sender,
                                    MouseEventArgs e);
```

- Event handler delegates use a signature that follows the event design pattern

Events

- Use strongly typed event data where appropriate

```
public class MouseEventArgs :
                              EventArgs { }
```

- Able to add new members without a breaking change

Events

- Provide a protected method to raise the event
  - Named On*EventName*
- Make it virtual if extensibility is needed

```
public class Button {
   private ButtonClickHandler onClickHandler;
   protected void OnClick (ClickEventArgs e) {
      if (onClickHandler != null) {
         // call the delegate if non-null
         onClickHandler(this, e);
      }
   }
}
```

Events

- Events are callbacks into arbitrary user code
  - Do not assume anything about the state of your object
  - Code defensively

```
protected void DoClick() {
   PaintDown(); // paint button in depressed state
   try {
      OnClick(); // call event handler
   }
   finally {
      // window may be deleted in event handler
      if (windowHandle != null) {
         PaintUp(); // paint button in normal state
      }
   }
}
```

Static Members

- Any kind of member can be static
- Static members
  - Cannot access instance state
  - Cannot override or specialize
  - Should be thread-safe
- Commonly used for
  - Factory methods
  - Singleton pattern
  - Utility methods

Static Members

- Statics are the Microsoft® .NET equivalent of global variables or global functions
  - Not object-oriented
  - Same evils as global
  - But can be very useful
    - System.Math—full modeling not required

```
public class Int32 {
  public static int Parse (string value) {..}
}
```

```
int i = Int32.Parse ("42");
```

Singleton Pattern

- Ensures that a class has only one instance and provides a global point of access to it
- This is not exactly the GoF pattern (see threading section)

```
public sealed class DBNull {
    private DBNull() {}
    public static readonly DBNull Value = new DBNull();
    // instance Methods...
}
```

```
if (x == DBNull.Value) {..}
```

Singleton Pattern

- Notice:
  - This class is sealed to prevent sub-classing to add instances
  - The **Value** is static for easy access
  - The **Value** is read-only so it cannot be modified
  - This class has a private constructor
  - The instance is immutable
    - No methods can mutate its state

Parameter Passing

- Value types and reference types can both be passed by value or by reference
- A value type by value copies the value
  - No side effects
  - Commonly used
    ```
    public int Add (int x, int y) {..}
    ```
- A value type by reference uses a pointer to the value
  - Side effects possible
  - Can be hard to use—often avoidable

```
public static int TryParse (string str,
  out int value)
```

Parameter Passing (continued)

- A reference type by value copied the reference
  - Side effects are possible on mutable types
  - Commonly used
    ```
    public void Insert (object value) {..}
    ```
- A reference type by reference uses a pointer to the reference variable
  - Side effects possible
  - Almost never used
    ```
    public static int Method (ref object moreData) {..}
    ```

Ref and Out Parameters

- Using Ref and Out parameters
  - Primarily used for interop
    - Avoid directly exposing publicly
  - May be used for extremely performance-sensitive areas
  - Almost exclusively used with value types
- Ref is a common language runtime (CLR) feature
  - Out is a C# feature
  - Out parameter semantics downgrade to Ref semantics in other languages

Ref, Out, and Value Params

- Ref and out designs are less usable

```
public void GetLocation (
          ref int x, out int y) {..}
```

```
int x = 42;
int y;
b.GetLocation (ref x, out y);
```

```
public struct Point {
  public int X {get;}
  public int Y {get;}
}
```

```
Point p = b.Location;
```

Argument Validation

- Do argument checking on every publicly exposed member
  - Catches errors early (fail-fast)
  - Much easier to debug
  - Is a powerful security precaution
- Throw meaningful exceptions
  - Subclasses of ArgumentExcoeption

Argument Checking

```
public int Count {
    get {return count;}
    set {
        if (value < 0 || value >= MaxValue)
            throw new ArgumentException(..);
    }
}
public void Select (int start, int end) {
    if (start < 0)
        throw new ArgumentException(..);
    if (end < start)
        throw new ArgumentException(..);
}
```

Exercise: Choose the Right Member

- A collection class needs a member to return the number of elements in the collection

Exercise: Choose the Right Member

- A data access class needs to express a user id, password, and connection string and a way to initiate a connection

Exercise: What's Wrong with This Type?

```
public class Student {
  public const int Retirement_Age = 60;
  public int Age;
  public string Getname () {}
  public Class this[int courseNumber]
       { get {...} }
  public void Register (string className) {
       Register (className, true);
  }
  public void Register (string className,
       bool waitList) {}
  public void Register (int courseNumber) {}
}
```

Exercise: What's Wrong with This Member?

```
public sealed class String {
   public static string Format (string formatString,
         object obj1) {
      return Format (formatString, obj1, null);
   }

   public static string Format (string formatString,
         object obj1, object obj2)  {
      return Format (formatString, obj1,
            obj2, null);
   }

   public static string Format (string formatString,
         object obj1, object obj2, object obj3) {
   // implementation
   ...
   }
}
```

Exercise: What's Wrong with This Member?

```
public class Convert {
    public string IntToString (int value) {}
    public string DoubleToString (double value)
    {}
}
```

Lesson 4 Summary

- Use the right member at the right time, in the right way
  - Constructors
  - Methods
  - Properties
  - Fields
  - Events